Memory Hierarchy and Caches

Computer Architecture

Riad Bourbia

Computer Sciences department

Guelma University

[Adapted from slides of Dr. A. El-maleh]

Presentation Outline

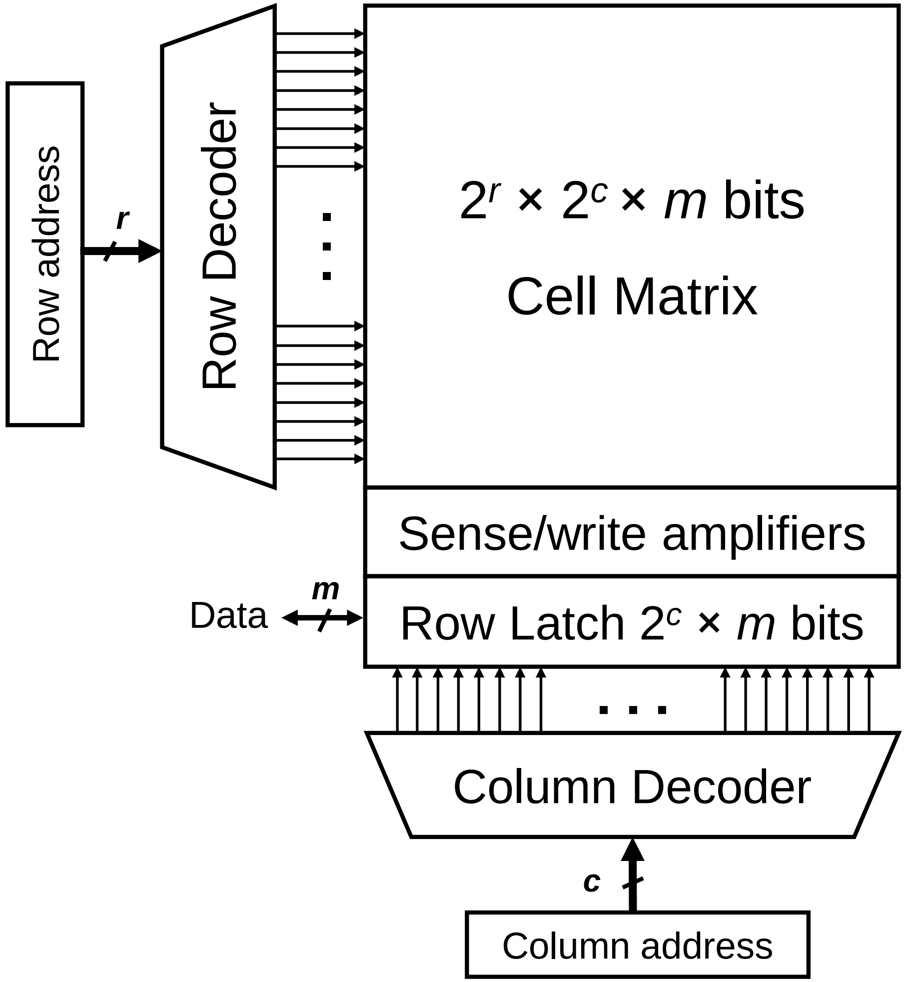
- **Random Access Memory and its Structure**
- Memory Hierarchy and the need for Cache Memory
- The Basics of Caches

Memory Technology

- Static RAM (SRAM)
  - Used typically to implement Cache memory
  - Requires 6 transistors per bit
  - Low power to retain bit
- Dynamic RAM (DRAM)
  - Used typically to implement Main Memory
  - One transistor + capacitor per bit
  - Must be re-written after being read
  - Must be refreshed periodically
    - By reading and rewriting all the rows in the DRAM

Typical Memory Structure

- Row decoder
  - Select row to read/write
- Column decoder
  - Select column to read/write
- Cell Matrix
  - 2D array of tiny memory cells
- Sense/Write amplifiers
  - Sense & amplify data on read
  - Drive bit line with data in on write
- Same data lines are used for data in/out



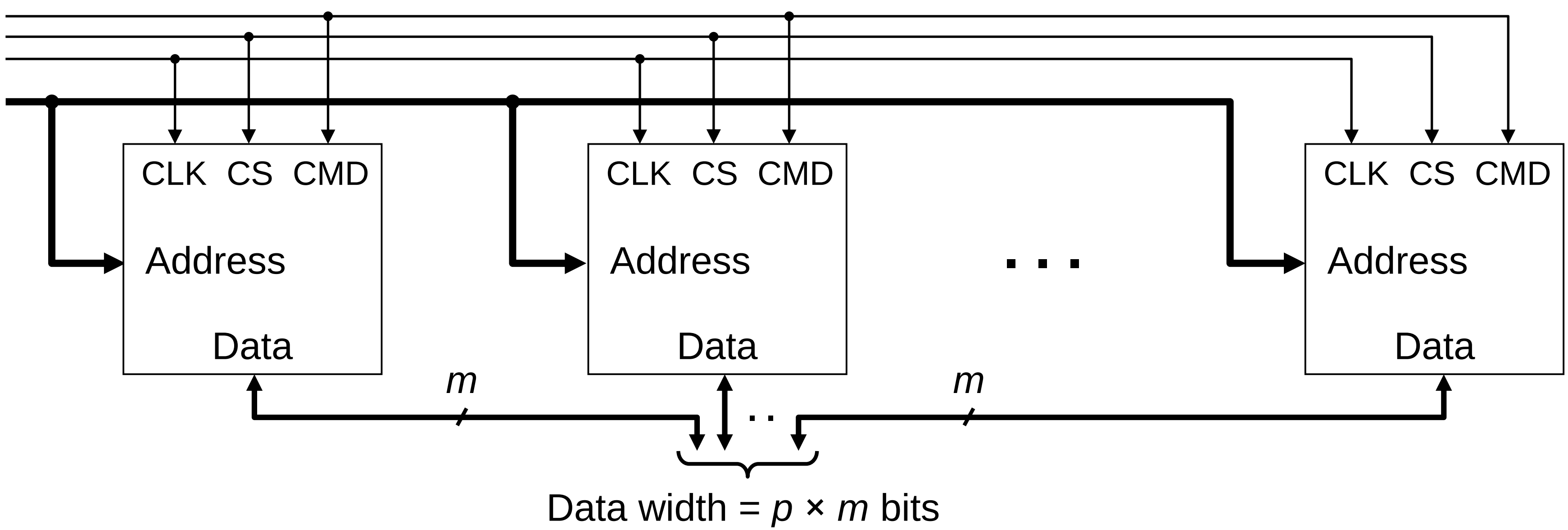
SDRAM and DDR SDRAM

- SDRAM is **Synchronous** Dynamic RAM
  - Added clock to DRAM interface
- SDRAM is synchronous with the system clock
  - Older DRAM technologies were asynchronous
  - As system bus clock improved, SDRAM delivered higher performance than asynchronous DRAM
- DDR is **Double Data Rate** SDRAM
  - Like SDRAM, DDR is synchronous with the system clock, but the difference is that DDR reads data on both the rising and falling edges of the clock signal

Memory Modules

❖ Memory Rank: Set of DRAM chips accessed in parallel

- ✧ Same Chip Select (CS) and Command (CMD)
- ✧ Same address, but different data lines
- ✧ Increases memory capacity and bandwidth
- ✧ Example: 64-bit data bus using 4 × 16-bit DRAM chips

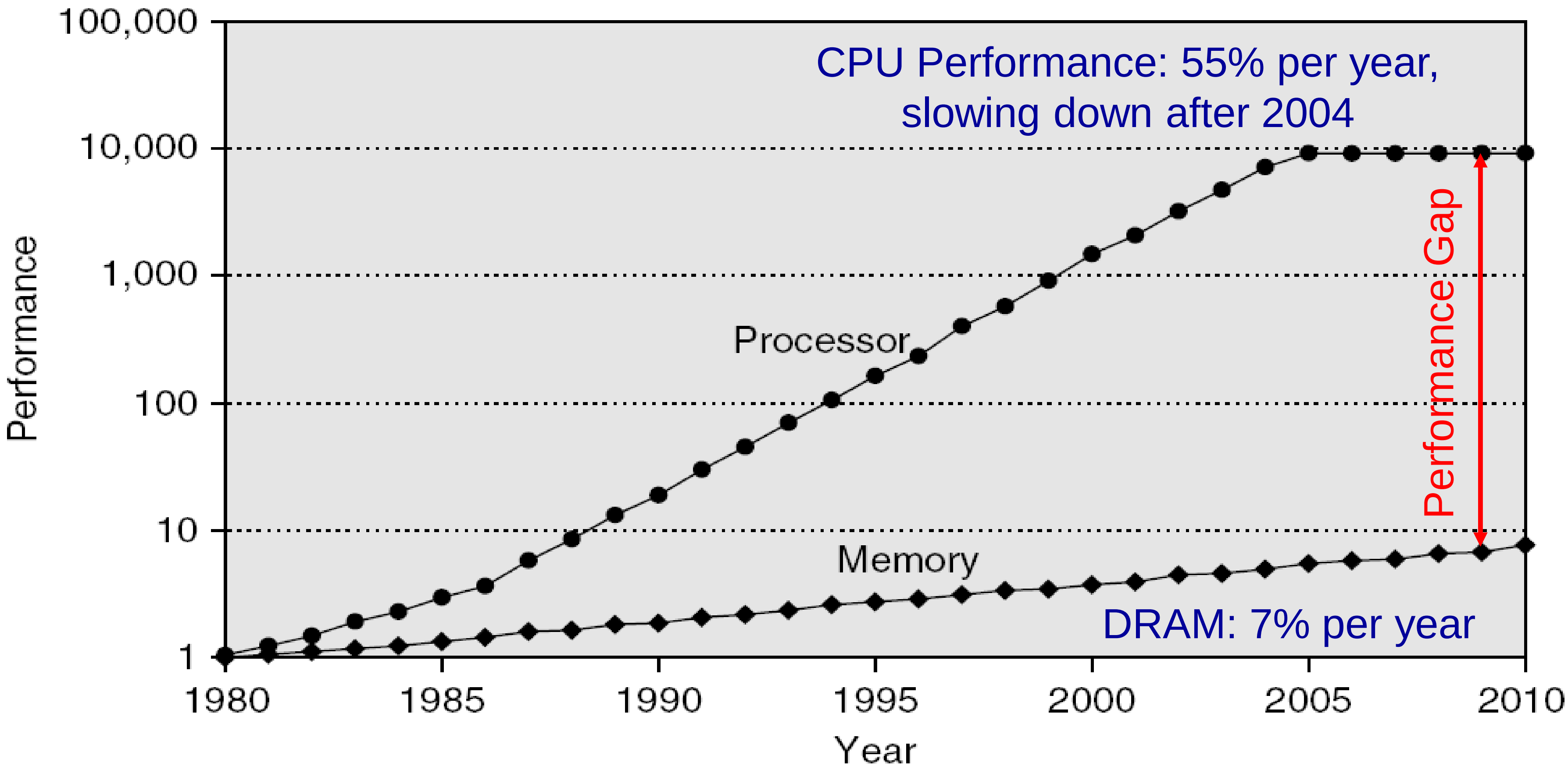


Trends in DRAM

Year	Memory Standard	Chip Capacity (Mibit)	Bus Clock (MHz)	Data Rate (MT/s)	Peak Bandwidth (MB/s)	Total latency to a new row / column
1996	SDRAM	64-128	100-166	100-166	800-1333	60 ns
2000	DDR	256-512	100-200	200-400	1600-3200	55 ns
2003	DDR2	512-2048	200-400	400-800	3200-6400	50 ns
2007	DDR3	2048-8192	400-800	800-1600	6400-12800	40 ns
2014	DDR4	8192-32768	800-1600	1600-3200	12800-25600	35 ns

- Memory chip capacity: 1 Mibit = 2^{20} bits, 1 Gibit = 2^{30} bits
- Data Rate = Millions of Transfers per second (MT/s)
- Data Rate = 2 × Bus Clock for DDR, DDR2, DDR3, DDR4
- 1 Transfer = 8 bytes of data ➔ Bandwidth = MT/s × 8 bytes

Processor-Memory Performance Gap



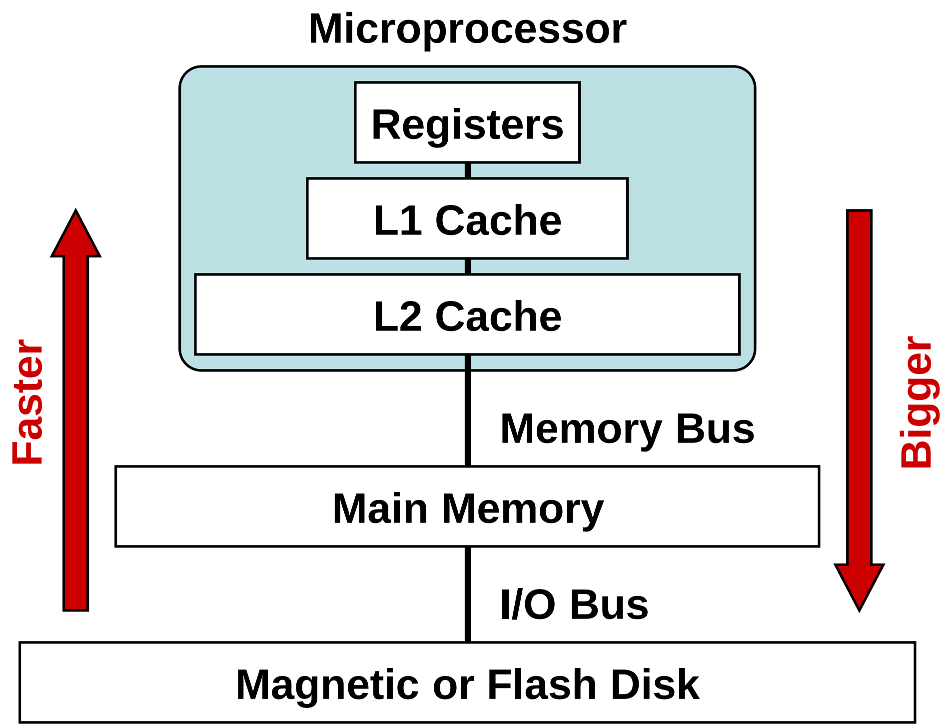
- 1980 – No cache in microprocessor
- 1995 – Two-level cache on microprocessor

The Need for Cache Memory

- Widening speed gap between CPU and main memory
  - Processor operation takes less than 1 ns
  - Main memory requires more than 50 ns to access
- Each instruction involves at least one memory access
  - One memory access to fetch the instruction
  - A second memory access for load and store instructions
- Memory bandwidth limits the instruction execution rate
- Cache memory can help bridge the CPU-memory gap
- Cache memory is small in size but fast

Typical Memory Hierarchy

- Registers are at the top of the hierarchy
  - Typical size < 1 KB
  - Access time < 0.5 ns
- Level 1 Cache (8 – 64 KiB)
  - Access time: 1 ns
- L2 Cache (1 MiB – 8 MiB)
  - Access time: 3 – 10 ns
- Main Memory (8 – 32 GiB)
  - Access time: 40 – 50 ns
- Disk Storage (> 200 GB)
  - Access time: 5 – 10 ms



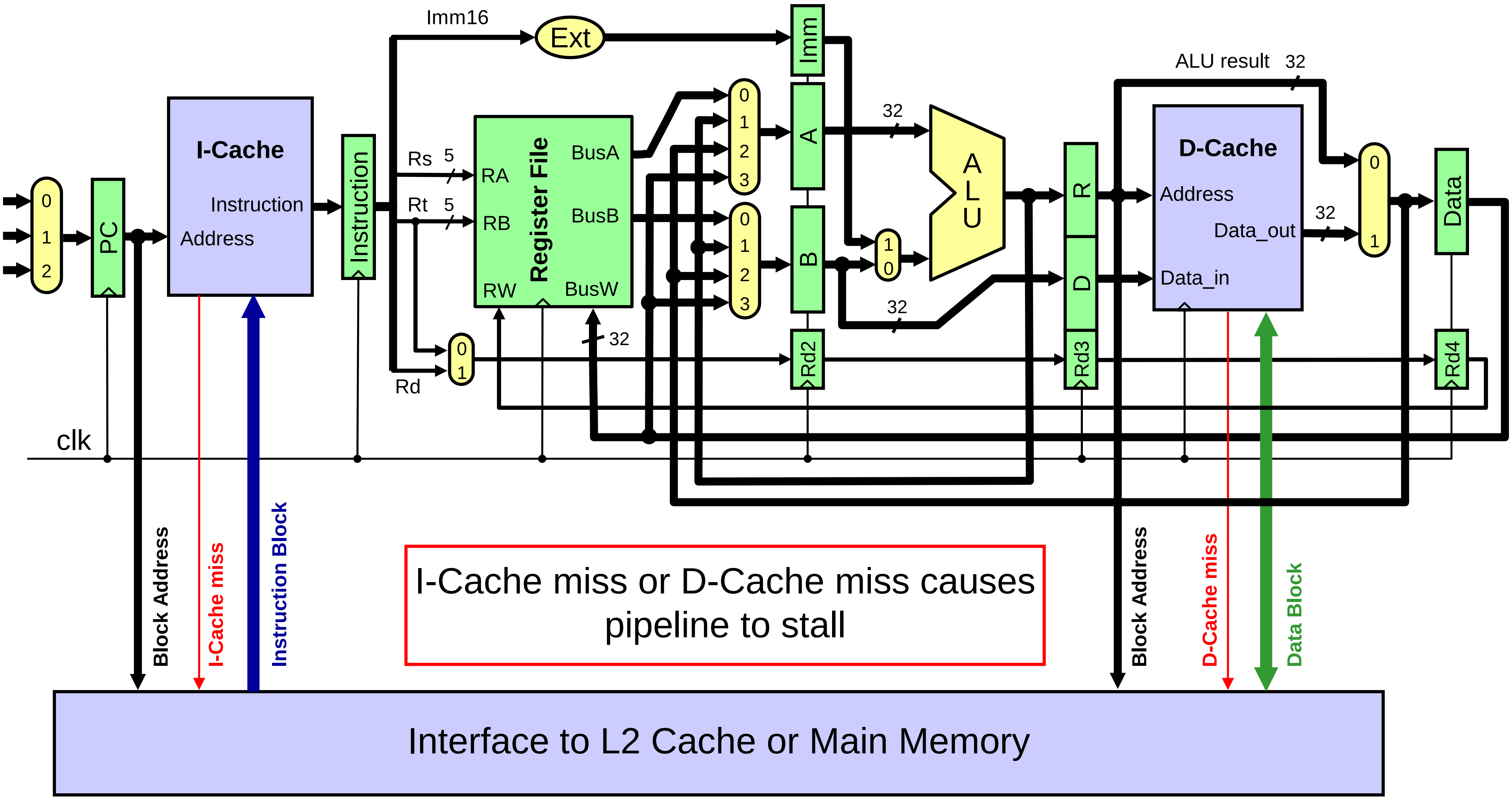
Principle of Locality of Reference

- Programs access small portion of their address space
  - At any time, only a small set of instructions & data is needed
- **Temporal Locality** (in time)
  - If an item is accessed, probably it will be accessed again soon
  - Same loop instructions are fetched each iteration
  - Same procedure may be called and executed many times
- **Spatial Locality** (in space)
  - Tendency to access contiguous instructions/data in memory
  - Sequential execution of Instructions
  - Traversing arrays element by element

What is a Cache Memory ?

- Small and fast (SRAM) memory technology
  - Stores the subset of instructions & data currently being accessed
- Used to reduce average access time to memory
- Caches exploit **temporal locality** by …
  - Keeping recently accessed data closer to the processor
- Caches exploit **spatial locality** by …
  - Moving blocks consisting of multiple contiguous words
- Goal is to achieve
  - **Fast speed** of cache memory access
  - Balance the **cost** of the memory system

Cache Memories in the Datapath



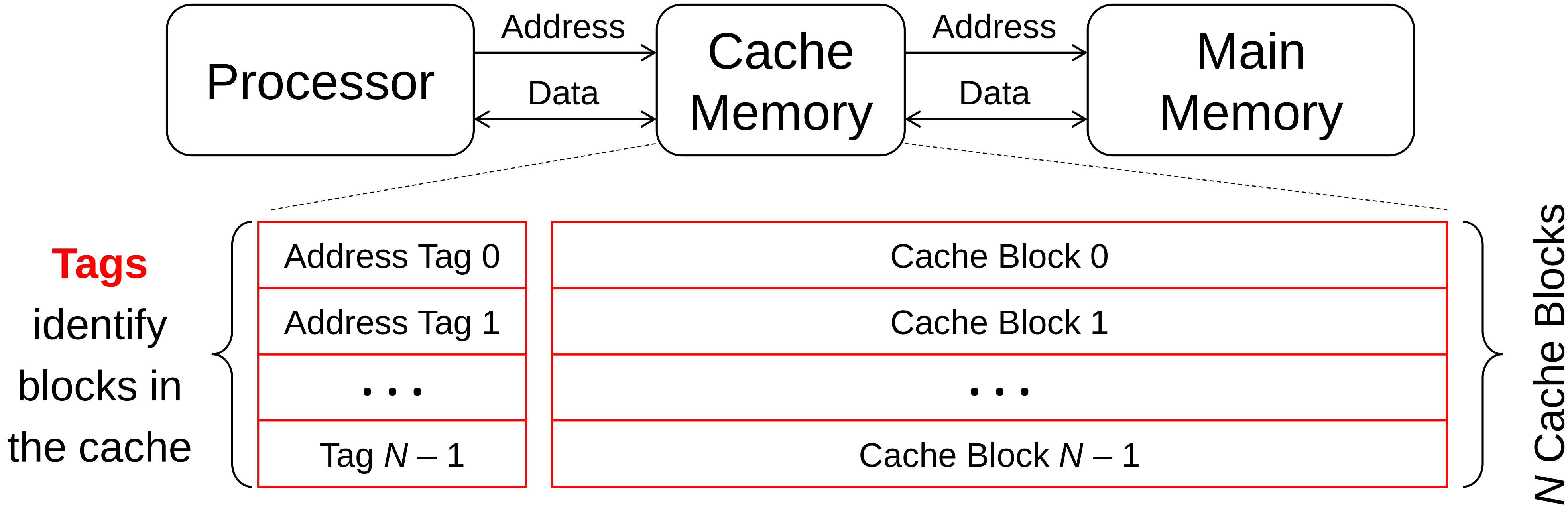
Almost Everything is a Cache !

- In computer architecture, almost everything is a cache!
- Registers: a **cache on variables** – software managed
- First-level cache: a **cache on second-level cache**
- Second-level cache: a **cache on memory** (or L3 cache)
- Memory: a **cache on hard disk**
  - Stores recent programs and their data
  - Hard disk can be viewed as an extension to main memory
- Branch target and prediction buffer
  - **Cache on branch target and prediction** information

Four Basic Questions on Caches

- Q1: Where can a block be placed in a cache?
  - **Block placement**
  - Direct Mapped, Set Associative, Fully Associative
- Q2: How is a block found in a cache?
  - **Block identification**
  - Block address, tag, index
- Q3: Which block should be replaced on a cache miss?
  - **Block replacement**
  - FIFO, Random, LRU
- Q4: What happens on a write?
  - **Write strategy**
  - Write Back or Write Through cache (with Write Buffer)

Inside a Cache Memory



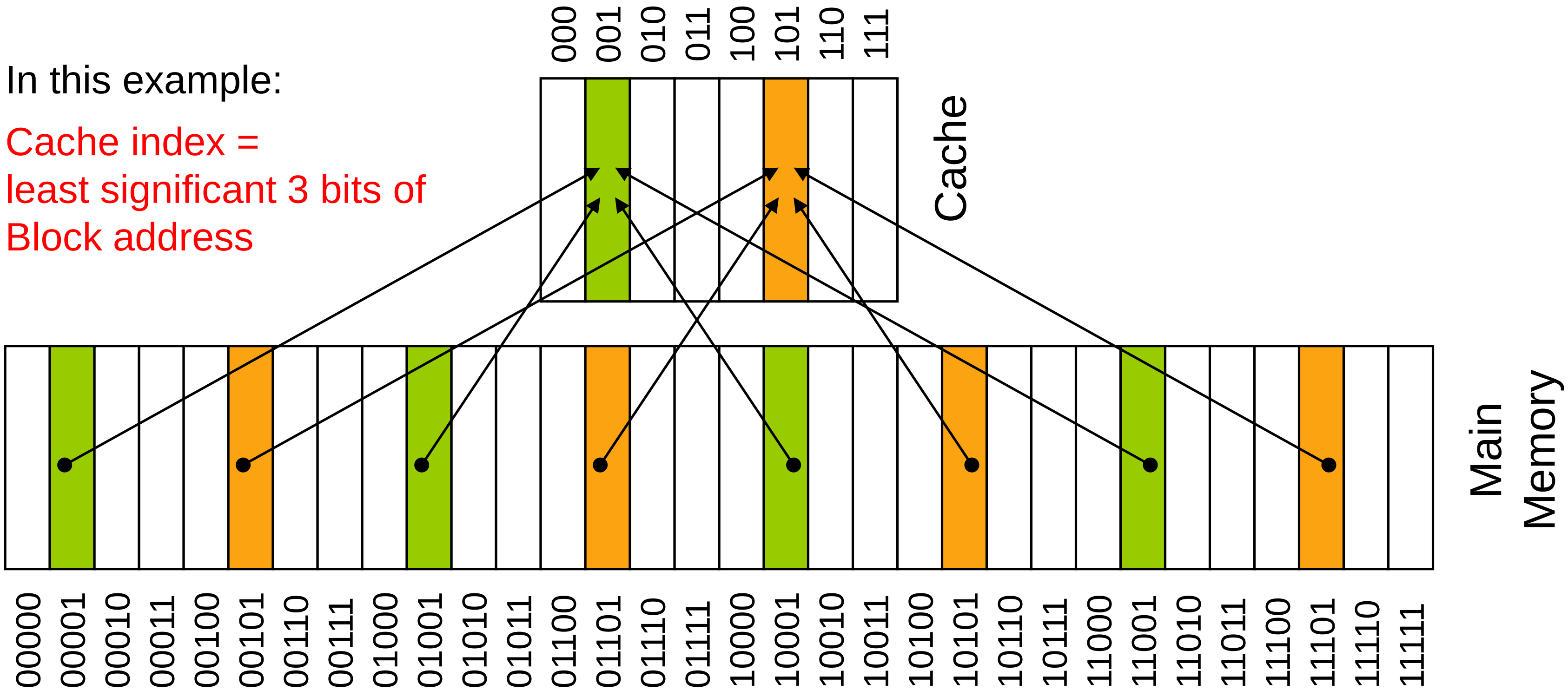
- **Cache Block** (or Cache Line)
  - Unit of data transfer between main memory and a cache
  - Large block size ➔ Less tag overhead + Burst transfer from DRAM
  - Typically, cache block size = 64 bytes in recent caches

Block Placement: Direct Mapped

- **Block**: unit of data transfer between cache and memory
- **Direct Mapped Cache**:
  - A block can be placed in exactly one location in the cache

In this example:

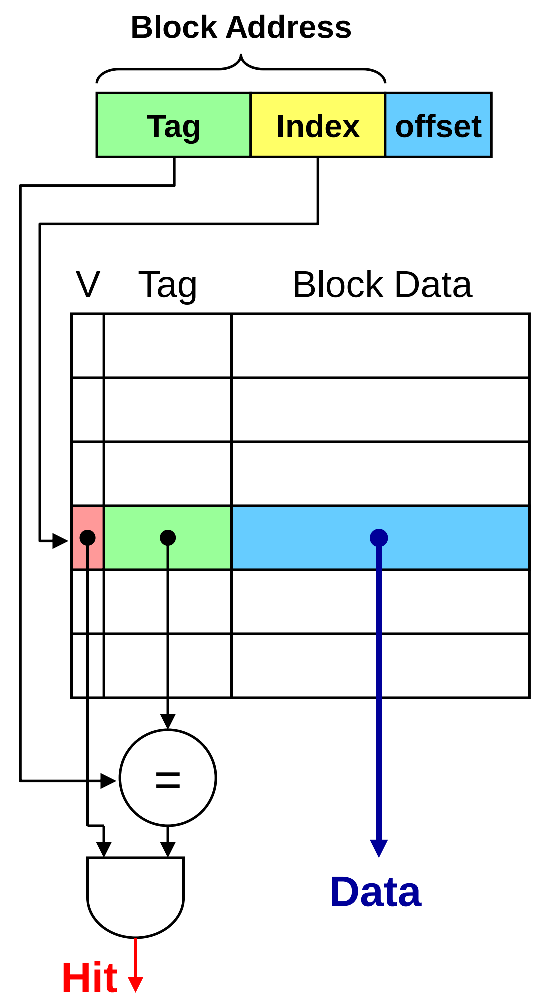
Cache index =
least significant 3 bits of
Block address



Direct-Mapped Cache

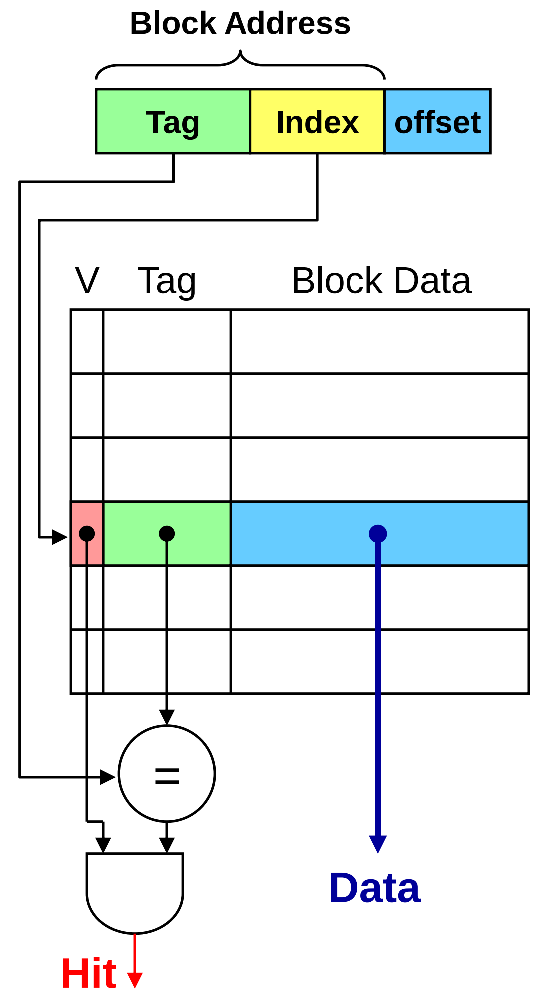
- A memory address is divided into
  - **Block address**: identifies block in memory
  - **Block offset**: to access bytes within a block
- A block address is further divided into
  - **Index**: used for direct cache access
  - **Tag**: most-significant bits of block address

  Index = Block Address **mod** *Cache Blocks*
- Tag must be stored also inside cache
  - For block identification
- A **valid bit** is also required to indicate
  - Whether a cache block is valid or not



Direct Mapped Cache – cont'd

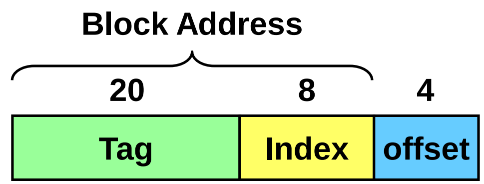
- **Cache hit**: block is stored inside cache
  - Index is used to access cache block
  - Address tag is compared against stored tag
  - If equal and cache block is valid then **hit**
  - Otherwise: **cache miss**
- If number of cache blocks is 2^n
  - n bits are used for the cache index
- If number of bytes in a block is 2^b
  - b bits are used for the block offset
- If 32 bits are used for an address
  - $32 - n - b$ bits are used for the tag
- Cache data size = 2^{n+b} bytes



Mapping an Address to a Cache Block

- Example
  - Consider a direct-mapped cache with 256 blocks
  - Block size = 16 bytes
  - Compute tag, index, and byte offset of address: 0x01FFF8AC

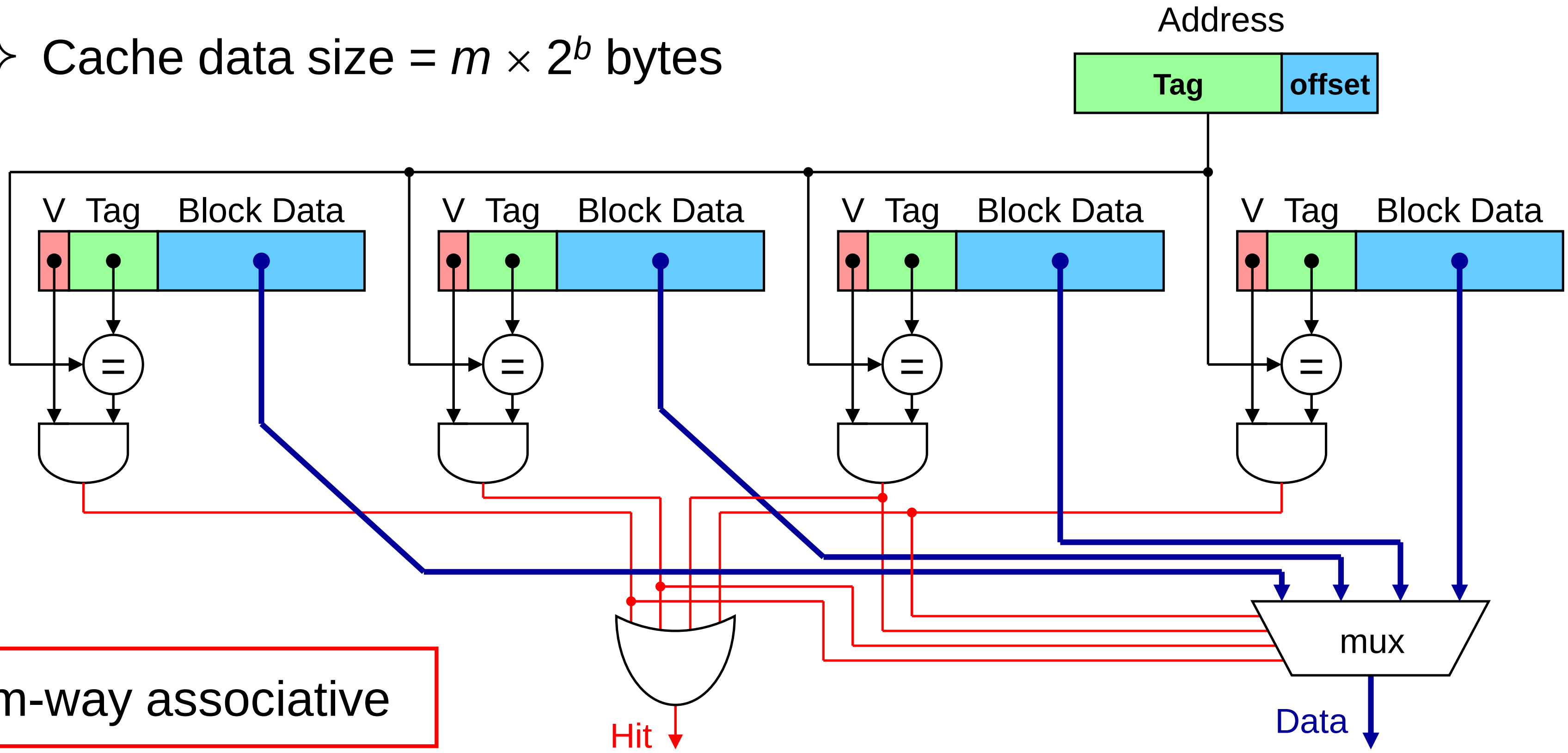
- **Solution**
  - 32-bit address is divided into:



    - 4-bit byte offset field, because block size = 2^4 = 16 bytes
    - 8-bit cache index, because there are 2^8 = 256 blocks in cache
    - 20-bit tag field
  - Byte offset = 0xC = 12 (least significant 4 bits of address)
  - Cache index = 0x8A = 138 (next lower 8 bits of address)
  - Tag = 0x01FFF (upper 20 bits of address)

Fully Associative Cache

- A block can be placed anywhere in cache ⇒ no indexing
- If *m* blocks exist then
  - *m* comparators are needed to match *tag*
  - Cache data size = $m \times 2^b$ bytes

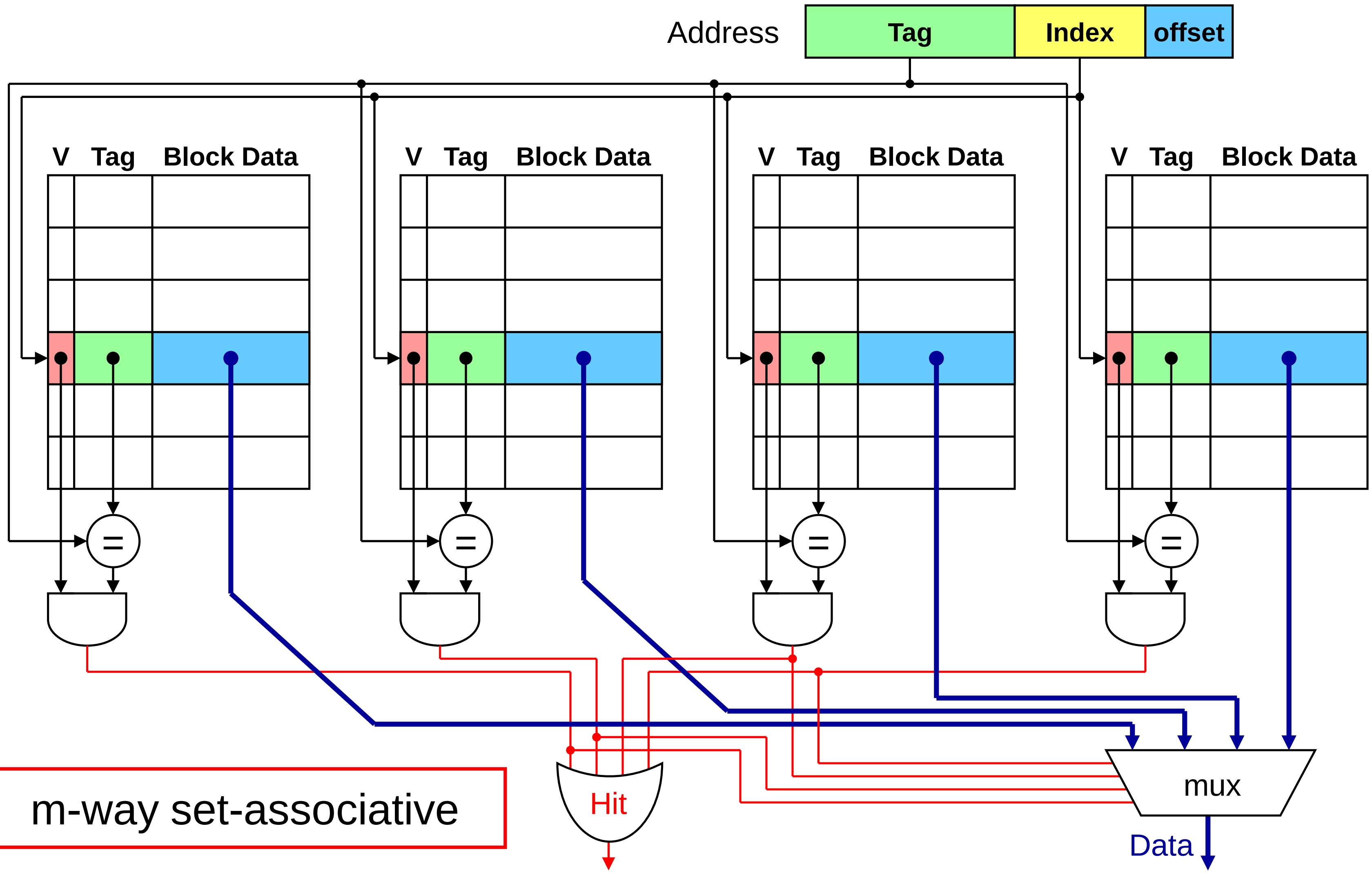


m-way associative

Set-Associative Cache

- A **set** is a group of blocks that can be indexed
- A block is first mapped onto a set
  - *Set index = Block address* **mod** *Number of sets in cache*
- If there are m blocks in a set (m-way set associative) then
  - m tags are checked in parallel using m comparators
- If 2^n sets exist then **set index** consists of n bits
- Cache data size = $m \times 2^{n+b}$ bytes (with 2^b bytes per block)
  - Without counting tags and valid bits
- A direct-mapped cache has one block per set (m = 1)
- A fully-associative cache has one set (2^n = 1 or n = 0)

Set-Associative Cache Diagram

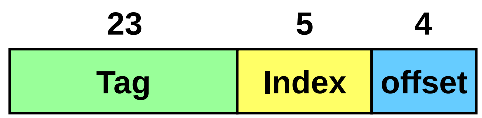


m-way set-associative

Example on Cache Placement & Misses

- Consider a small direct-mapped cache with 32 blocks
  - Cache is initially empty, Block size = 16 bytes
  - The following memory addresses (in decimal) are referenced:

    1000, 1004, 1008, 2548, 2552, 2556.
  - Map addresses to cache blocks and indicate whether hit or miss
- Solution:



  - 1000 = 0x3E8 cache index = 0x1E Miss (first access)
  - 1004 = 0x3EC cache index = 0x1E Hit
  - 1008 = 0x3F0 cache index = 0x1F Miss (first access)
  - 2548 = 0x9F4 cache index = 0x1F Miss (different tag)
  - 2552 = 0x9F8 cache index = 0x1F Hit
  - 2556 = 0x9FC cache index = 0x1F Hit

Write Policy

- **Write Through:**
  - Writes update cache and lower-level memory
  - Cache control bit: only a Valid bit is needed
  - Memory always has latest data, which simplifies data coherency
  - Can always discard cached data when a block is replaced
- **Write Back:**
  - Writes update cache only
  - Cache control bits: Valid and Modified bits are required
  - Modified cached data is written back to memory when replaced
  - Multiple writes to a cache block require only one write to memory
  - Uses less memory bandwidth than write-through and less power
  - However, more complex to implement than write through

What Happens on a Cache Miss?

- Cache sends a **miss signal** to **stall** the processor
- Decide which cache block to allocate/replace
  - One choice only when the cache is directly mapped
  - Multiple choices for set-associative or fully-associative cache
- Transfer the block from lower level memory to this cache
  - Set the valid bit and the tag field from the upper address bits
- If block to be replaced is **modified** then write it back
  - Modified block is written back to memory
  - Otherwise, block to be replaced can be simply discarded
- Restart the instruction that caused the cache miss
- **Miss Penalty:** clock cycles to process a cache miss

Replacement Policy

- Which block to be replaced on a cache miss?
- No selection alternatives for direct-mapped caches
- *m* blocks per set to choose from for associative caches
- **Random replacement**
  - Candidate blocks are randomly selected
  - **One counter for all sets** (0 to $m - 1$): incremented on every cycle
  - On a cache miss replace block specified by counter
- **First In First Out (FIFO) replacement**
  - Replace oldest block in set
  - **One counter per set** (0 to $m - 1$): specifies **oldest block** to replace
  - Counter is incremented on a cache miss

Replacement Policy – cont'd

- **Least Recently Used (LRU)**
  - Replace block that has been **unused for the longest time**
  - Order blocks within a set from least to most recently used
  - Update ordering of blocks on each cache hit
  - With *m* blocks per set, there are ***m*!** possible permutations
- Pure LRU **is too costly** to implement when *m* > 2
  - *m* = 2, there are 2 permutations only (a single bit is needed)
  - m = 4, there are 4! = 24 possible permutations
  - LRU approximation is used in practice
- For large *m* > 4,

  Random replacement can be as effective as LRU